### 1. ****Explain CORS enablement for Web API access for local application****

CORS, or **Cross-Origin Resource Sharing**, is a mechanism that enables a web application running at one origin (like http://localhost:3000) to access resources hosted on another origin (like http://localhost:5000). Without enabling CORS, modern browsers block cross-origin requests by default for security reasons. In this project, CORS was enabled in the Program.cs file by adding a named policy using builder.Services.AddCors(), which allowed any origin, method, and header using .AllowAnyOrigin(), .AllowAnyMethod(), and .AllowAnyHeader(). This policy was then activated with app.UseCors("AllowLocal") to ensure local frontend apps or tools like Postman could access the Web API without CORS-related issues.

### 2. ****Demonstrate security in WebAPI using Bearer and JWT token authentication****

This project implemented **JWT (JSON Web Token)** based authentication to protect Web API endpoints. A JWT token is a compact, URL-safe token that contains claims such as the user ID and role. In Program.cs, the API was configured to use AddAuthentication() and AddJwtBearer() with validation parameters like issuer, audience, signing key, and token lifetime. A long symmetric security key was used to meet encryption standards. The AuthController was created with [AllowAnonymous], which allowed unauthenticated access to generate a token. Inside it, the GenerateJSONWebToken() method was implemented using the JwtSecurityTokenHandler, which returned a token containing role and user ID claims valid for 2 minutes. This token could be copied and used in request headers to access secured endpoints.

### ****Using [Authorize] attribute and Claims-based Role Authorization****

Once the JWT token was generated from the /api/Auth endpoint, secured access to the EmployeeController was enforced using the [Authorize] attribute. To implement **role-based authorization**, [Authorize(Roles = "Admin,POC")] was used on the controller. This ensured that only users with the Admin or POC roles (embedded in the JWT claims) could access the endpoints. If the token was missing, expired, or had the wrong role, the API responded with a 401 Unauthorized status code. Postman was used to test this behavior by including the token in the Authorization header using the format: Bearer <token>. If the token matched and had the correct role, a 200 OK was returned; otherwise, access was denied.

### ****Check for JWT expiration and test expired tokens****

To demonstrate JWT expiration, the expires parameter in GenerateJSONWebToken() was set to 2 minutes. This short expiration time helped simulate real-world security behavior. After generating the token and waiting for more than 2 minutes, attempts to access a secured API like /api/Employee with the expired token resulted in 401 Unauthorized, proving that the backend successfully validated token expiry. This feature enhances security by ensuring that tokens are only valid for a short duration unless renewed or refreshed by the client.